Table Lookup Improvement

Use less memory

Mon, Jul 8, 2002

The scheme introduced recently to eliminate network table searches, as much as possible, has one unfortunate feature: it needs 640K bytes of memory. This note explores how the scheme can be modified to reduce the memory requirements.

The basic scheme uses 512K bytes of memory for a table that has 64K entries of 8 bytes each, which is enough to supply 4 node numbers for all possible IP addresses for a Class B internet. In practice, one is not dealing with more than a tiny fraction of that number of IP addresses. Looked at in terms of Fermilab 256-address subnets, nodes from only 6–8 subnets may be in use at one time. But we could also look at the situation with 64-address subnets. Right now, `node0509` uses 12 subnets of this size out of 81 active entries. Checking `node0600`, the Linac data server node, there are 14 subnets of this size out of 116 active entries.

What would the situation look like with 64-address subnets? Each subnet can be supported with 64 entries of 8 bytes each, for 512 bytes per subnet. But we would like a table lookup method to find such a 512-byte subnet table. A table of 2-byte subnet indices would suffice, allowing for a separate index for every possible 64-address subnet in a Class B internet. With 1024 such indices, it would require 2048 bytes. If we assumed a maximum of 32 possible 64-address subnets may be supported at one time, we need a memory space of 32*512 + 2048 = 18K bytes. This is a major reduction from 512K bytes.

The above discussion covers the IPNOD table. What about the related IPADD table? It now uses 128K bytes, since it has 32K entries of 4-byte IP addresses. But this table is also sparsely populated. It needs to be addressed, in rough terms, by indices in ranges `0x0500—0x07FF`, `0x0900—0x10FF`, and `0x6000—0x6FFF`. The sizes of these three ranges are, respectively, `0x0300`, `0x0800`, and `0x1000`, or `0x1B00` total, which is about 7K. The total space needed would be about 28K bytes, which is much less than the present 128K bytes. But a further modification can be made. The `0x6000—0x6FFF` range only uses one in every 16 entries, so it could be further mapped to `0x0000—0x00FF`. With this change, we need only range sizes of `0x0300`, `0x0800`, and `0x0100`, for a total of `0x0C00` entries, or 12K bytes.

But we must consider all known node IP addresses, not only the ones that are reflected in a typical snapshot of the IPARP table. Scanning a relatively full IPNAT, there are 27 such subnets represented. Scanning the TRUNK tables, there may be 40 or so. This indicates that a better maximum number of subnets supported would be 64, say, rather than 32. This would increase the size of memory needed for the IPNOD function to 34K bytes.

The memory needed for both tables is then 12K + 34K = 46K bytes. If we reserved 64K bytes, allowing for 100 subnets at once (100*512 + 2048 = 52K), that would still be ten times less than was used in the first implementation.

What are the details of the table indexed by 64-address subnet number? Each block of 512 bytes that supports one such subnet can be indexed from 0–99, if we consider 100 possible subnets. If we place this block index number in a 2-byte word scaled at 2^9, then it has the same appearance as the offset to the block. For example, if we refer to block 1 as `0x0200`, it would represent 512 bytes, which is the correct offset to block 1. Block 99 would be represented as `0xC600` in this scheme. The other 9 bits in the 2-byte value may be used for a count of active nodes in that subnet. This count could range from 0–64 decimal, of course.

Given an IP address, what should be done to map to an appropriate index into a new IPNOD

structure? Assuming it is in the local Class B range, check the 10-bit subnet number. Using it as an index into this new `SubBlkB` array, if the word found there is zero, it means that there is no assigned area in use for this subnet. The first job is to allocate one. How can it be done?

An array of 100 bytes, `BlockC`, could hold a count of entries in use for each possible block. An index into this array is the block number in the range 0–99. If the byte is zero, it means that block is free. To allocate a new block for use with a new subnet, one can scan through this array looking for a vacant block, starting at the block that had last been found in this way. When a zero byte is found, the index to that byte, left-shifted 9 bits, represents the offset to be stored in the `SubBlkB` table. The count in the `BlockC` array should be set to 1, and the index to that byte should be used to update the record of the last found empty byte, which will be used as a starting point the next time a new subnet scan is required.

All this was done to allocate a new subnet block. The need for this arose because a nonzero node number was to be assigned to a nonzero IP address via a Set routine call, and it was an IP address in a subnet that had no associated block yet. If there was a nonzero offset found in the `SubBlkB` array entry, then it is simply the offset to be used, along with the 6-bit subnet index (the low 6 bits of the IP address) to reach the 8-byte entry containing the 4 possible node numbers.

If the node number is zero, then it is possible that storing the zero into the relevant field of the entry will cause the 8-byte entry to be clear, in which case the count associated with that block can be decreased. If decreasing the count of active node entries produces a zero count, then it means that the subnet block is no longer needed, so its `BlockC` entry can be cleared. (Of course, decreasing the count, which is what the `BlockC` contains, has already cleared it.) If all is working correctly, the subnet block itself should be free and clear, since all 8-byte entries are known to be zero.

Do we need a count in the `SubBlkB` table entry, or is the offset enough? It appears that the count would have only a diagnostic value, indicating that the related subnet of addresses includes a specific number of nonzero 8-byte entries in its subnet block. If this count can be dispensed with, the table of 1024 entries could be only a table of 1024 bytes. If convenient, the block number times 2 could be stored here, so as to more easily relate it to the block offset.

Going beyond local Class B

Can we devise a scheme that can also support rapid lookup of any IP address, even though it is not part of the local Class B internet? Suppose there is first a check against the local Class B internet range. If there is a match, continue as described above. Failing to find a match, a short search through a list of dynamically-allocated subnets can be made. Imagine a table with entries that hold a 26-bit subnet and a block number. If this search fails, the set routine logic can allocate another such entry and allocate a free block count, just as is done for the usual case. Again, each such block can support up to 64 node addresses that share the same upper 26 bits. The short table of IP addresses might be called `Subnet`. A parallel table of byte entries to hold block numbers might be called `SubBlkX`.

How can all these structures be packaged? Suppose the objective is to build a 64K-byte structure altogether. The components needed are the new IPADD and IPNOD, the `SubBlkB`, the `SubBlkX`, the `BlockC`, the `ExtSub`, plus miscellaneous diagnostics. A possible fixed address for the entire 64K byte structure is `0x001E0000`.

Offset	*Size*	*Component*
`0`	12K	IPADD (see below)
`0x003000`	64	Header (see below)

`0x003040`	64	`SubBlkX`: analogous to `SubBlkB`, but for external subnets
`0x003080`	128	`BlockC`: array of counts of entries in use within subnet block
`0x003100`	256	`ExtSub`: list of 64 possible external 26-bit subnets
`0x003200`	512	`SubDiag`: diagnostic log of subnet allocations
`0x003400`	1K	`SubBlkB`: block numbers for Class B 64-address subnets in use
`0x003800`	50K	IPNOD subnet blocks (100 blocks of 512 bytes each)
`0x010000`	(64K)	(total size)

This allocation places the various arrays on boundaries that are multiples of their sizes. Note that IPNOD, which has 100 blocks of 512 bytes each, ends the entire 64K byte block. The breakdown of the IPADD portion, which consists of 3 parts, is as follows:

Offset	*Size*	*Component*
`0`	1K	pseudoN: 256 entries for pseudo node#s
`0x0400`	3K	nativeN: 768 entries for native node#s
`0x1000`	8K	acnetN: 2048 entries for 8 trunks of 256 Acnet node#s each
`0x3000`	(12K)	(total size)

The 64-byte header of the entire NODES table, found at offset `0x3000`:

Offset	*Size*	*Field*
0	2	`nativeNN`: copy of native node number
2	2	`acnetNN`: copy of Acnet node number
4	4	`localIP`: copy of local IP address from IPARP table
8	2	`fullCnt`: count of number of times table limits exceeded
10	2	`blockCN`: last found `BlockC` entry
12	2	`extSubN`: length of `ExtSub` array
14	2	`nBlocks`: number of subnet blocks allocated
16	4	`uHdrPtr`: pointer to user space in data stream queue
20	2	`uHdrSz`: size of user space in data stream queue
22	2	`subDiagN`: `SubDiag` array index for next record

The 16-byte SubDiag entries might look like this:

Offset	*Size*	*Field*
0	4	`subDIPA`: IP address causing subnet block allocate/free
4	2	`subDNod`: node# being set into subnet block entry
6	2	`subDBlk`: subnet block# with hi 4 bits A(allocated) or F(freed)
8	8	`subDTim`: Date-time of allocation.

When a block becomes free, because the last entry in use has become 8 bytes of zeros, it is necessary to retire the `SubBlkB` entry, for the case of the Class B group, or to retire the `SubBlkX` group, for the case of the external subnets in `ExtSub`. In order to do this, the address of the byte in `BlockC` whose count was just cleared must be considered. Its index is the block number, which was just found by one of two means, either in `SubBlkB` or `SubBlkX`. The byte in the appropriate entry must be cleared to show that there is no longer an associated subnet block in use. The next time that same block is needed, a new block will be allocated. For the external case, when the `ExtSub` entry is cleared, make sure that any holes are cleared. This can be used to maintain a counter in the header of how many ExtSub entries need be searched. The size of the table allows for 64 external subnets, but we only want to search the entries that are in use. Most of the time, there may be only the multicast block here, unless that is handled in a special way.

The allocation business is handled by IPMAP. But for `GetxNode` routines, it is not necessary to allocate a subnet block if it does not currently exist.